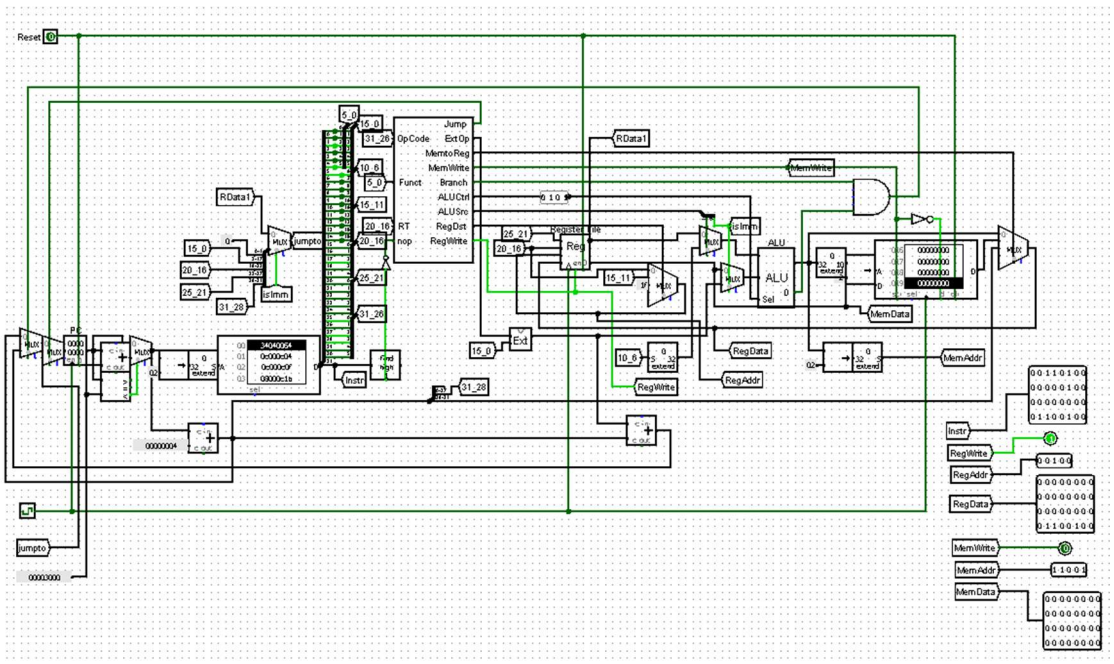
Verilog 开发 MIPS 单周期处理器

一、整体结构：

控制器（Controller）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）、EXT（位扩展器）。

处理器为 32 位处理器

处理器应支持的指令集为：{addu, subu, ori, lw, sw, beq, lui, jal,jr,nop}。



二、数据通路设计：

（一）模块规格撰写

1. IFU（取指令单元）。包括 PC（程序计数器，5 位），IM（指令存储器，32 位*32 字）

为了和 Mars 汇编器保持一致，PC 需要初始化为 0x30000000，因此，采取如下方法：

当 reset 信号为 1 时初始化为 0，加上 0x3000 作为 IM 的地址。之后，如果 PC 值大于等于 0x3000 则直接作为地址，否则加上 0x3000。可以证明，这个有限状态机中的值永远不小于 0x3000，但保证地址不重复加 0x3000。

1）PC（程序计数器）

器件：32bit 寄存器

模块接口

信号名	方向	描述
next[31:0]	I	下一条指令的地址
clk	I	时钟信号
reset	I	复位信号 1：有效 0：无效
reg[31:0]	O	当前指令地址（IAddr=32'h00003000）

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，PC 被设置为起始地址 0x00003000

2）IM（指令存储器）

IM 容量为 4KB（32bit×1024 字）

因为 ROM 中储存了 1024 个地址，且 IM 实际地址宽度仅为 10 位，，从而将地址的低 10 位（2~11 位）连接到 ROM 选择地址端口。

模块接口

信号名	方向	描述
RAddr[9:0]	I	输入当前的地址
RData[31:0]	O	输出读取当前的数据

2. Controller（控制器）

模块接口

信号名	方向	描述
cmd[31:0]	I	32 位操作编码
Jump	O	跳转信号 0 为不是跳转指令 1 为是跳转指令
RegSrc[1:0]	O	寄存器地址的来源
MemWrite	O	DM 写控制信号，写入 GRF 的数据选择(内存写使能信号)
Branch	O	分支信号 输出 0 为不是 Branch 输出为 1 是 Branch
ALUSrc[1:0]	O	ALU 操作数 2 的来源
RegDst[1:0]	O	寄存器地址选择 0:[20:16] 1:[15:11]
RegWrite	O	寄存器写使能信号
ExtOp[1:0]	O	控制扩展方式
ALUCtrl[3:0]	O	ALU 功能选择信号

3.GRF（通用寄存器组）：内部包括 32 个寄存器

具有写使能的寄存器实现，寄存器总数为 32 个 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0

模块接口

信号名	方向	描述
IAddr[31:0]	I	相应指令存储地址
clk	I	时钟信号
reset	I	复位信号 1：有效 0：无效
WEnable	I	读写控制信号 1：写操作 0：读操作
RAddr1[4:0]	I	读寄存器 1 的地址
RAddr2[4:0]	I	读寄存器 2 的地址
WAddr[4:0]	I	5 为地址输入信号，指定 32 个寄存器中的一个作为写入目标寄存器地址
WData[31:0]	I	向写寄存器中写入的值（数据）
RData1[31:0]	O	32 位输出 1
RData2[31:0]	O	32 位输出 2

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，所有寄存器的值被设置为 0x00000000
2	写寄存器	根据输入的写寄存器地址，把输入的数据写入写寄存器中
3	读寄存器	根据输入的读寄存器地址，将数据读出

4.ALU（算术逻辑单元）

提供 32 位加、减、或运算可以不支持溢出

模块接口

信号名	方向	描述
op1[31:0]	I	ALU32 位输入数据 A

op2[31:0]	I	ALU32 位输入数据 B
sel[3:0]	I	ALU 功能选择信号
Result[31:0]	O	32 位数据输出
Zero	O	输出为 0

5.EXT（位扩展器）：

模块接口

信号名	方向	描述
imm[15:0]	I	16 位 imm 数据输入
EOp[1:0]	I	位扩展选择信号 00：高位符号扩展 01：高位补 0 10：低位补 0 11：符号扩展之后，左移两位
ext[31:0]	O	位扩展后的 32 位输出

模块接口（extbyte）

信号名	方向	描述
imm[7:0]	I	8 位 imm 数据输入
EOp	I	位扩展选择信号 0：高位补 0 1：低位补 0
ext[31:0]	O	位扩展后的 32 位输出

6.DM（数据存储器）

DM（数据存储器，32 位*1024 字）。用内置 RAM 实现，采用 Separate load and store ports 属性。地址 10 位。题目中要求输出 5 位地址，取后 5 位。RAM 自带时钟信号、写使能、地址端和数据端，与 DM 要求完全相同。

起始地址：0x00003000

模块接口

信号名	方向	描述
clk	I	时钟信号
WE	I	读写控制信号 1：写操作 0：读操作

reset	I	复位信号 1：有效 0：无效
isu	I	判断是否无符号或有符号数
MemDst[1:0]	I	写入数据的输入
Addr[11:0]	I	读寄存器的地址
WData[31:0]	I	向写寄存器中写入的值（数据）
IAddr[31:0]	I	相应指令存储地址
RData[31:0]	O	32 位输出

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，所有数据被设置为 0x00000000
2	写操作	根据输入的寄存器地址，把输入的数据写入
3	读操作	根据输入的寄存器地址，将其中的数据读出

（二）思考题

1. 根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

答：在 DM 中，内存是按字划分的。一个字是四字节，因此在进行 DM 中进行内存的存取时，应当将输入地址除以 4（或者右移两位），作为实际地址，即取 2~11 位。

addr 信号来自 ALU 的运算结果。

2. 在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

答：reset 针对 PC、GRF、DM，其中 PC 需要回到初始指令地址 0x00003000，GRF 需要清空全部寄存器的值，DM 也清空。

（三）IM 设计

```
module Instr_Memory(
input [9:0]RAddr,
output [31:0]RData
);
        reg [31:0] rom[0:1023];
         integer i;
         initial begin
                for(i=0;i<1024;i=i+1) rom[i]=0;
                $readmemh("code.txt",rom);
        end
        assign RData=rom[RAddr];
endmodule
```

（四）控制器设计

（一）设计方式

1. 数据通路设计

PC：输入端接 next 信号，输出端接指令存储器。

IM：输出端接指令译码器。

DM：写数据端接寄存器文件输出，读数据端接寄存器文件写数据。写地址端接 ALU 结果。MemWrite 接写使能端。

ALU：操作数接寄存器和立即数，输出端接寄存器写数据和内存写地址。还有一个零端，与 Branch 信号接在与门上，当二者均为真时分支有效。

next 信号（下一个 PC 值）：当 j 为真时，为立即数。否则当 branch 和 zero 均为真时，为 branch 计算所得地址；

2. 主控单元真值表

指令	Opcode	Funct	Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
addu	000000	100001	0	xx	0	0	0	0010	0	1	1
subu	000000	100011	0	xx	0	0	0	0011	0	1	1
ori	001101	xxxxxx	0	01	0	0	0	0101	1	0	1
lw	100011		0	00	1	0	0	0010	1	0	1
sw	101011		0	00	0	1	0	0010	1	x	0
beq	000100		0	11	0	0	1	0011	0	x	0
lui	001111		0	10	0	0	0	0010	0	0	1

j	000010		1	00	x	0	x	xxxx	x	x	0

（二）思考题

1. 列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。

答：

1）利用 if-else（或 case）完成操作码和控制信号的值之间的对应；

```
always@(*)begin
  case(op)
   6'b000000:
      begin
  if(func==6'b001000)begin          //jr
       RegDst=2'b00;
        ALUSrc=0;
        MemtoReg=2'b00;
        RegWrite=0;
        MemWrite=0;
        PCSrc=2'b00;
        Extop=2'b00;
        ALUop=2'b00;
        jr=1;
        beq=0;
  end
  else if(func==6'b100001)begin      //addu
 RegDst=2'b01;
  ALUSrc=0;
  MemtoReg=2'b00;
  RegWrite=1;
  MemWrite=0;
  PCSrc=2'b00;
  Extop=2'b00;
  ALUop=2'b00;
  jr=0;
  beq=0;
```

```
end
else if(func==6'b100011)begin      //subu
      RegDst=2'b01;
       ALUSrc=0;
       MemtoReg=2'b00;
       RegWrite=1;
       MemWrite=0;
       PCSrc=2'b00;
       Extop=2'b00;
       ALUop=2'b01;
            jr=0;
            beq=0;
end
    else begin
             RegDst=2'b00;
             ALUSrc=0;
             MemtoReg=2'b00;
             RegWrite=0;
             MemWrite=0;
             PCSrc=2'b00;
             Extop=2'b00;
             ALUop=2'b00;
             jr=0;
                beq=0;
                end
            end
```

2）利用 assign 语句完成操作码和控制信号的值之间的对应；

```
assign RegDst[0] = ( (op == 6'b000000 && func == 6'b100001) || (op == 6'b000000 && func == 6'b100011) )? 1 : 0 ;
```

3）利用宏定义

```
`define state1 4'b0001
`define state2 4'b0010
`define state3 4'b0100
```

`define state4 4'b1000

`define 标识符(宏名) 字符串(宏内容)

如：`define signal string

它的作用是指定用标识符 signal 来代替 string 这个字符串，在编译预处理时，把程序中在该命令以后所有的 signal 都替换成 string。这种方法使用户能以一个简单的名字代替一个长的字符串，也可以用一个有含义的名字来代替没有含义的数字和符号，因此把这个标识符(名字)称为"宏名"，在编译预处理时将宏名替换成字符串的过程称为"宏展开"。`define 是宏定义命令。

2. 根据你所列举的编码方式，说明他们的优缺点。

编码方式	优点	缺点
if-else/case	代码写起来比较容易、直观，与 C 语言类似	它们在不注意的情况下容易产生锁存器，对毛刺敏感，使其处于不确定状态，且语句较为冗长
assign	语句较为简单，看起来比较方便	适用范围比较窄，只能对 wire 型变量赋值
宏定义	增加可读性、可修改性好、设计的可重用性好	定义时容易出错，且不容易查错

（五）在线测试相关信息

（一）测试代码

```
.data

arr:.space 44

.text

ori $t0,$0,0

ori $t1,$0,1

ori $s0,$0,1

ori $s1,$0,4

ori $s2,$0,40

for:

beq $t0,$s2,next

sw $t1,arr($t0)

addu $t1,$t1,$s0

addu $t0,$t0,$s1

j for
```

```
next:

ori $t0,$0,0

ori $t1,$0,0

ori $s0,$0,0

for1:

beq $t0,$s2,next1

lw $t1,arr($t0)

addu $s0,$s0,$t1

addu $t0,$t0,$s1

j for1

next1:

sw $s0,arr($t0)

li $t0,0x44897253
```

导出的机器码为

```
34080000 34090001 34100001 34110004

34120028 11120004 ad090000 01304821

01114021 08000c05 34080000 34090000

34100000 11120004 8d090000 02098021

01114021 08000c0d ad100000 3c014489

34287253
```

（二）预期结果

Code	Basic
0x34080000	ori $8, $0, 0x00000000
0x34090001	ori $9, $0, 0x00000001
0x34100001	ori $16, $0, 0x00000001
0x34110004	ori $17, $0, 0x00000004
0x34120028	ori $18, $0, 0x00000028
0x11120004	beq $8, $18, 0x00000004
0xad090000	sw $9, 0x00000000($8)
0x01304821	addu $9, $9, $16
0x01114021	addu $8, $8, $17
0x08000c05	j 0x00003014
0x34080000	ori $8, $0, 0x00000000
0x34090000	ori $9, $0, 0x00000000
0x34100000	ori $16, $0, 0x00000000
0x11120004	beq $8, $18, 0x00000004
0x8d090000	lw $9, 0x00000000($8)
0x02098021	addu $16, $16, $9
0x01114021	addu $8, $8, $17
0x08000c0d	j 0x00003034
0xad100000	sw $16, 0x00000000($8)
0x3c014489	lui $1, 0x00004489
0x34287253	ori $8, $1, 0x00007253

（三）思考题

1.C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分 。

答：addi 和 addiu 的区别只是 addi 在发生溢出时会报错，add 与 addu 也是一样，在发生溢出时 add 会报错。所以在不考虑溢出的情况下，addi 和 add 忽略报错，因此等价。

ADDI：符号加立即数

编码	31 26	25 21	20 16	15 0
	addi 001000	rs	rt	immediate
	6	5	5	16
格式	addi rt, rs, immediate			
描述	GPR[rt] ← GPR[rs]+ immediate			

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + sign_extend(immediate)
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rt] ← temp$_{31..0}$
	endif			

示例	addi $s1, $s2, -1			
其他	temp$_{32}$ ≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 addi 与 addiu 等价。			

ADDIU：无符号加立即数

编码	31 26	25 21	20 16	15 0
	addiu 001001	rs	rt	immediate

	6	5	5	16
格式	addiu rt, rs, immediate			
描述	GPR[rt] ← GPR[rs]+ immediate			
操作	GPR[rt] ← GPR[rs] + sign_extend(immediate)			
示例	addiu $s1, $s2, 0x3FFF			
其他	“无符号”是一个误导，其本意是不考虑溢出。			

ADD：符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	add 100000
	6	5	5	5	5	6
格式	add rd, rs, rt					
描述	GPR[rd] ← GPR[rs]+GPR[rt]					

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + (GPR[rt]$_{31}$‖GPR[rt])
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rd] ← temp$_{31..0}$
	endif					

示例	add $s1, $s2, $s3					
其他	temp$_{32}$≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 add 与 addu 等价。					

ADDU：无符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	addu 100001
	6	5	5	5	5	6
格式	addu rd, rs, rt					
描述	GPR[rd] ← GPR[rs] + GPR[rt]					
操作	GPR[rd] ← GPR[rs] + GPR[rt]					
示例	addu $s1, $s2, $s3					
其他						

2. 根据自己的设计说明单周期处理器的优缺点。

答：

优点	缺点
控制部件相比多周期 CPU 更简单，且实际设计起来较为简便，不容易出错	用一个时钟周期执行一条指令，从而确定时钟周期的时间长度要考虑执行的时间最长的指令，一次确定 CPU 频率，不管指令复杂度如何，单周期 CPU 花费相同时间执行，这造成时间上浪费

简要说明 jal、jr 和堆栈的关系

答：在跳转到指定地址实现子程序调用的同时，需要将返回地址保存到 ra 寄存器，即通常所说的“函数调用的现场保护”，以便子程序返回时能够继续调用之前的流程。对于跳转/分支指令，MIPS CPU 将自动保存 ra；若子程序需要嵌套调用其他子程序，则必须先存储 ra，通常是压入栈，子程序末尾弹出之前保存的 ra，然后 jr 到 ra。这两条指令分别实现了直接和间接子程序调用。